Trabalho Prático 3

Consultas em sistema de escalonamento logístico de transportes

Mateus Antinossi Cordeiro Queiroz – 2024023767

Departamento de Ciência da Computação (DCC)

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

mateusacq@ufmg.br

1. Introdução

O problema proposto foi a implementação de um programa que, munido de algoritmos e estruturas de dados, lesse informações de pacotes, armazéns e transportes que uma empresa administrava com um sistema caracterizado por "Hanoi" (empilhamento de pacotes). O sistema processa os dados de entrada linha a linha, intercalando o tratamento de eventos e a resposta imediata às consultas CL (por cliente) e PC (por pacote), conforme o tempo dos eventos.

Esta documentação se subdivide nas seguintes seções:

- **2. Método:** seção que trata da implementação: estruturas de dados, tipos abstratos de dados e métodos implementados;
- **3. Análise de Complexidade:** seção que analisa a complexidade de tempo e de espaço dos elementos apresentados na seção 2, utilizando notação assintótica;
- **4. Estratégias de Robustez:** seção onde são discutidas a programação defensiva e a capacidade do código de resistir a falhas e prosseguir com a execução;
- **5. Análise Experimental:** avaliação de diferentes dimensões que impactam o custo computacional da execução do código.
- **6. Conclusão.**

2. Método

O programa foi desenvolvido em C++, usando o paradigma de orientação a objetos para ler os eventos logísticos. As classes principais são: **Arvore**, **Evento** e **ListaEventos**, além das estruturas de suporte: Heap e Lista.

A leitura da entrada é feita via linha de comando:
./tp3.out <arquivo_de_entrada.txt>

2.1. main():

Basicamente, lia-se do arquivo de entrada uma série de movimentações feitas com pacotes pela empresa em questão ao longo do tempo. Então, algumas consultas à respeito dos dados fornecidos eram feitas. As consultas poderiam ser do tipo “CL” (retorna os eventos iniciais e finais dos pacotes que um cliente possuía) e “PC” (retorna o histórico de eventos de um determinado pacote).

2.2. TAD Evento:

Essa classe representava um evento que poderia ser de 6 tipos: 0 (registro), 1 (armazenamento), 2 (remoção), 3 (rearmazenamento), 4 (transporte) e 5 (entrega) (mostrados na tabela). Eles foram enumerados com uma struct “enum”, própria da linguagem C++. Essa distinção era crucial para armazená-los na memória e facilitar as consultas deles.

Atributos	RG	AR	RM	UR	TR	EN
Data Hora	1	1	1	1	1	1
Tipo evento	3	3	3	3	3	3
ID Pacote	4	4	4	4	4	4
Remetente	5					
Destinatário	6					
Armazém Origem	7				5	
Armazém Destino	8	5	5	5	6	5
Seção Destino		6	6	6		

Figura 1. Tabela com tipos de eventos e seus atributos

2.3. Lista:

Implementação da estrutura de dados lista encadeada com um nó cabeça sentinela.

Possui apontadores para a primeira célula (célula apontada pelo nó cabeça), e para a última. Conta com getters e setters para a primeira posição e para um item em alguma posição passada no método e com os métodos:

- **Posiciona() e PosicionaAnterior():** métodos internos usados para obter um apontador para uma célula em alguma posição.
- **Insere e remove Início, final ou Posição:** métodos para inserir ou remover células no início (primeira célula após o nó cabeça), no final (apontado por “último”) ou em uma posição passada na função.

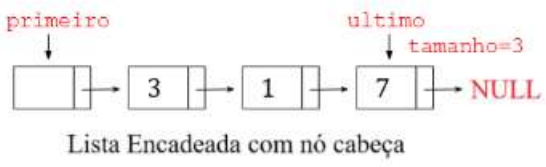

Figura 2. Esquema ilustrativo da estrutura de dados lista encadeada

2.6 Árvore AVL:

Foi implementada uma árvore binária de pesquisa com pseudobalanceamento, do tipo AVL. Essa estrutura foi usada para facilitar as consultas, tendo sido instanciada para representar dois índices principais: pacotes e clientes. No primeiro, eram obtidos facilmente os eventos associados a um pacote, pendurados em uma lista na sua célula correspondente. No segundo, os identificadores de pacotes e o tempo em que foram registrados estavam associados aos clientes que os possuíam, facilitando a busca por eles da maneira que a consulta “CL” pedia.

2.5 ListaEventos:

Classe filha de “Lista” que herdava seus métodos, aproveitando o princípio da herança do paradigma de programação orientada a objetos. Aqui um método especial para inserir eventos foi implementado, para que ele se adequasse ao tipo de célula e tipo de item que o template da lista instanciava para esse caso.

Foi usada como item da árvore que permitia a busca pelo índice de pacotes. Para maior eficiência, essa era a única estrutura que continha os eventos e, para evitar duplicatas, armazenava ponteiros para eles.

2.6. Heap:

Fila de prioridade que organiza os eventos pelo tempo, implementada como um **min-heap**. O heap possuía atributos onde eram definidos o tamanho e a capacidade dele, juntamente de um vetor de elementos do tipo Evento, que representava a estrutura. Possuía métodos de inserção e remoção da raiz (InsereEvento() e RetiraProximoEvento()) e métodos internos que garantiam suas propriedades de Heap (ordenação de pais menores que filhos), são eles: **GetAncestral(), GetSucessorDir(), GetSucessorEsq(), HeapifyPorBaixo() e HeapifyPorCima()** (recursivos).

Foi usada para imprimir, na ordem cronológica, os eventos iniciais e finais obtidos para determinados pacotes na consulta “CL”.

3. Análise de Complexidade

Nessa seção, será feita uma análise da complexidade de tempo e de espaço dos métodos implementados, formalizada pela notação assintótica.

3.1 Tempo

3.1.1 O(1)

Getters, setters, inserção/remoção simples nas estruturas lista e heap são métodos com complexidade de tempo O(1), já que a lista é encadeada e o heap sofre constante manutenção.

3.1.2 O(n)

Considerando “n” o número de elementos na estrutura: os métodos que limpam as estruturas de dados: (Limpa() e destrutores) e os métodos de impressão dos elementos possuem complexidade O(n).

3.1.3 $O(\log_2 n)$:

As funções de “Heapify” e o método de inserção no Heap possuem, no pior caso, complexidade $O(\log_2 n)$, que seria o caso em que percorreriam a altura do heap, caminho este que possui complexidade $\log_2 n$. O mesmo vale para os métodos de inserção, remoção e pesquisa na árvore AVL.

3.1.4 main()

Considerando p = pacotes, c = clientes, k = pacotes por cliente:

3.1.4.1 Comando “EV” (adição de novos eventos):

Se evento é de registro (RG):

- Busca e inserção no índice de pacotes: O(log p);
- Busca e inserção no índice de clientes: O(log c) para localizar cliente e O(log k) para inserir novo pacote na árvore pendurada. Feita 2 vezes (uma para o remetente, outra para o destinatário).

Total: O(log p + 2 * log c + 2 * log k).
Se evento é qualquer outro:

- Busca no índice de pacotes: O(log p);
- Inserção na lista de eventos: O(1);

Total: O(log p) + O(1) = max(O(log p), O(1)) = O(log p).

3.1.4.2 Comando “CL” (consulta de pacotes do cliente)

- Busca do cliente: O(log c);
- Cópia profunda da árvore pendurda: O(k*log k)
- Busca cada um dos k pacotes no índice de pacotes: O(k log p);
- Acessos aos 2k eventos inicial e final: O(1);
- Insere nos escalonadores de inicio e fim: O(log k), k vezes em cada;
- Impressão final: O(k).

Total: O(log c + k log k + k log p).

3.1.4.3 Comando “PC” (consulta eventos do pacote)

- Busca no índice de pacotes: O(log p);
- Supondo e eventos, a impressão da lista de eventos possui complexidade de tempo O(e).

Total: O(max(log p, e)).

3.2 Espaço

Usando as mesmas variáveis da seção 3.1.4 e ainda no main():

3.2.1. Heap:

O(p), no pior caso.

3.2.2. IndicePacotes:

Árvore AVL com até P pacotes: O(p). Cada pacote possui uma ListaEventos com até e eventos: complexidade fica, então, O(p * e).

3.2.3 IndiceClientes:

Árvore AVL com até c clientes: O(c). Cada cliente possui uma AVL com até k pacotes: complexidade fica, então, O(c * k).

3.2.4 Complexidade de espaço total

Combinando os índices e o heap: O(p*e + c*k).

3.3 Conclusão

A análise confirma que a decisão de fazer um sistema de consultas com estruturas de dados com balanceamento (AVL e Heap as principais) foi ótima. Por causa disso, as inserções e buscas feitas garantem complexidade logarítmica de tempo e espaço.

4. Estratégias de Robustez

Para garantir robustez e evitar falhas durante a execução, foram aplicadas as seguintes estratégias:

4.1. Validação da entrada

O programa garante que a linha de comando correta é usada na sua execução pelo terminal, sem comprometer o resto da execução, através de uma verificação no começo do main().

4.2. Monitoramento e depuração

Nas estruturas de dados, há o lançamento de erros com o método std::cerr e a verificação de itens nulos para notificar comportamentos inesperados dos métodos. Mensagens de erro detalhadas acompanham operações e movimentações importantes do programa.

Essa estratégia facilita a depuração do programa e o tratamento de falhas, contribuindo, também, para a robustez do código.

4.3. Controle de memória

Estratégias que prevenissem segmentation fault foram usadas com a ideia de controlar memória. Uma delas envolvia a verificação dos valores passados em getters e setters, para ver se estavam dentro dos limites comportados pela estrutura.

Além disso, todas as classes com alocação dinâmica possuem construtor de cópia, destrutor e sobrecarga do operador de atribuição, respeitando a Rule of Three e prevenindo shallow copies, double free e dangling pointer.

5. Análise Experimental

3 experimentos analisaram variações em dimensões da entrada. Para tanto, um gerador de cargas de eventos de trabalho aleatórios com os parâmetros de interesse controlados, foi usado. A partir dele, uma base com os dados retornados foi usada para plotar gráficos que os exibissem.

5.1 Metodologia

O tempo foi medido usando a biblioteca <chrono> em C++. Os dados foram coletados através do uso de um script run.csh para gerar a entrada e um script em python para ler a saída. A partir dos dados obtidos, gráficos foram gerados usando matplotlib.pyplot. O número de consultas de clientes e pacotes foi mantido constante em 5.

5.2 Resultados

5.2.1. Variação do número de eventos pela topologia

A variação desse parâmetro reduziu significativamente o tempo de execução total, já que os pacotes estavam mais distribuídos e poderiam ser gerenciados de maneira “diluída”, ao passo que o tempo médio de consultas por cliente aumentou muito aqui. Esse segundo comportamento era mais inesperado, no entanto, provavelmente ocorreu assim por causa da busca mais trabalhosa por pacotes em eventos mais variados no armazém que estavam.

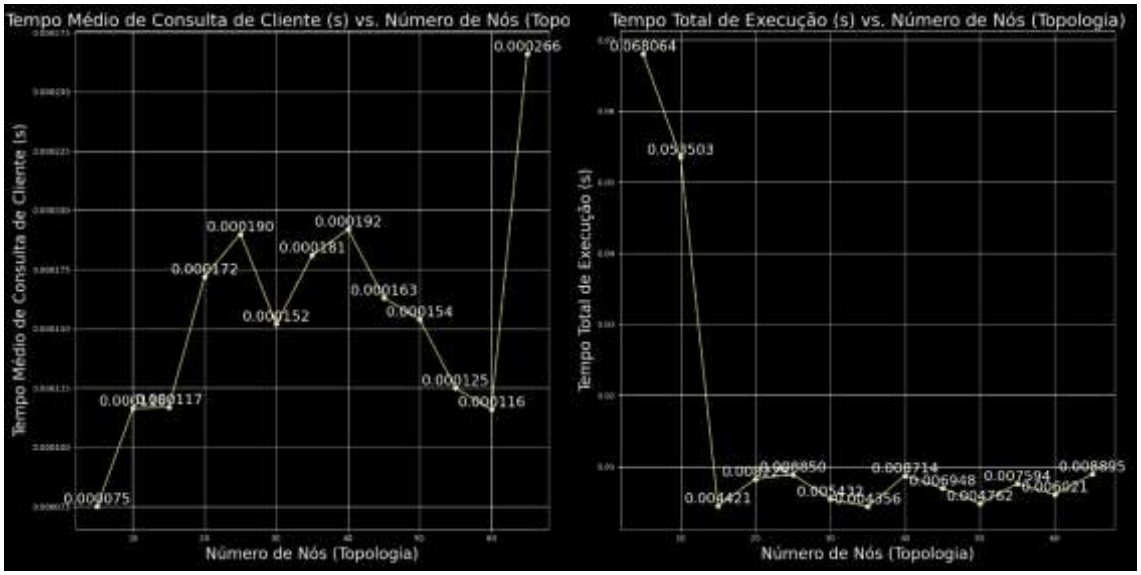

Figura 3. Gráficos com os tempos de execução médio da consulta CL e total nesse experimento

5.2.2 Variação do número de eventos pela contenção do transporte (intervalo entre transportes)

O tempo de execução total aumentou mais com essa variação, provavelmente porque ela causa gargalos no sistema. A curva parece ser logarítmica, análoga ao crescimento assintótico da função de complexidade de tempo do programa.

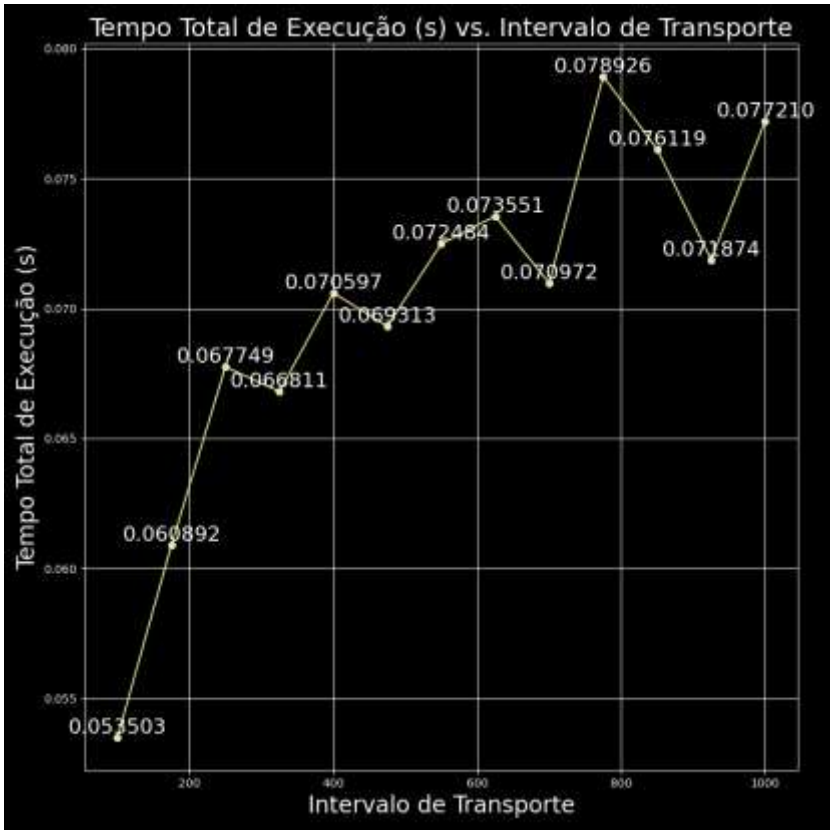

Figura 4. Relação entre o aumento do intervalo entre transportes e o tempo total de execução

5.2.3 Variação do número de pacotes

Conforme o número de pacotes crescia, o tempo de execução de todo o programa, bem como o tempo médio de cada consulta CL crescia também. Isso era esperado, já que o maior número de pacotes está diretamente relacionado a uma maior carga de trabalho do sistema logístico e um número maior de eventos armazenados para clientes, ao passo que as consultas por pacote não foram muito influenciadas (pequena variação de 1e-5 a 6e-5), uma vez que são feitas para cada pacote individualmente.

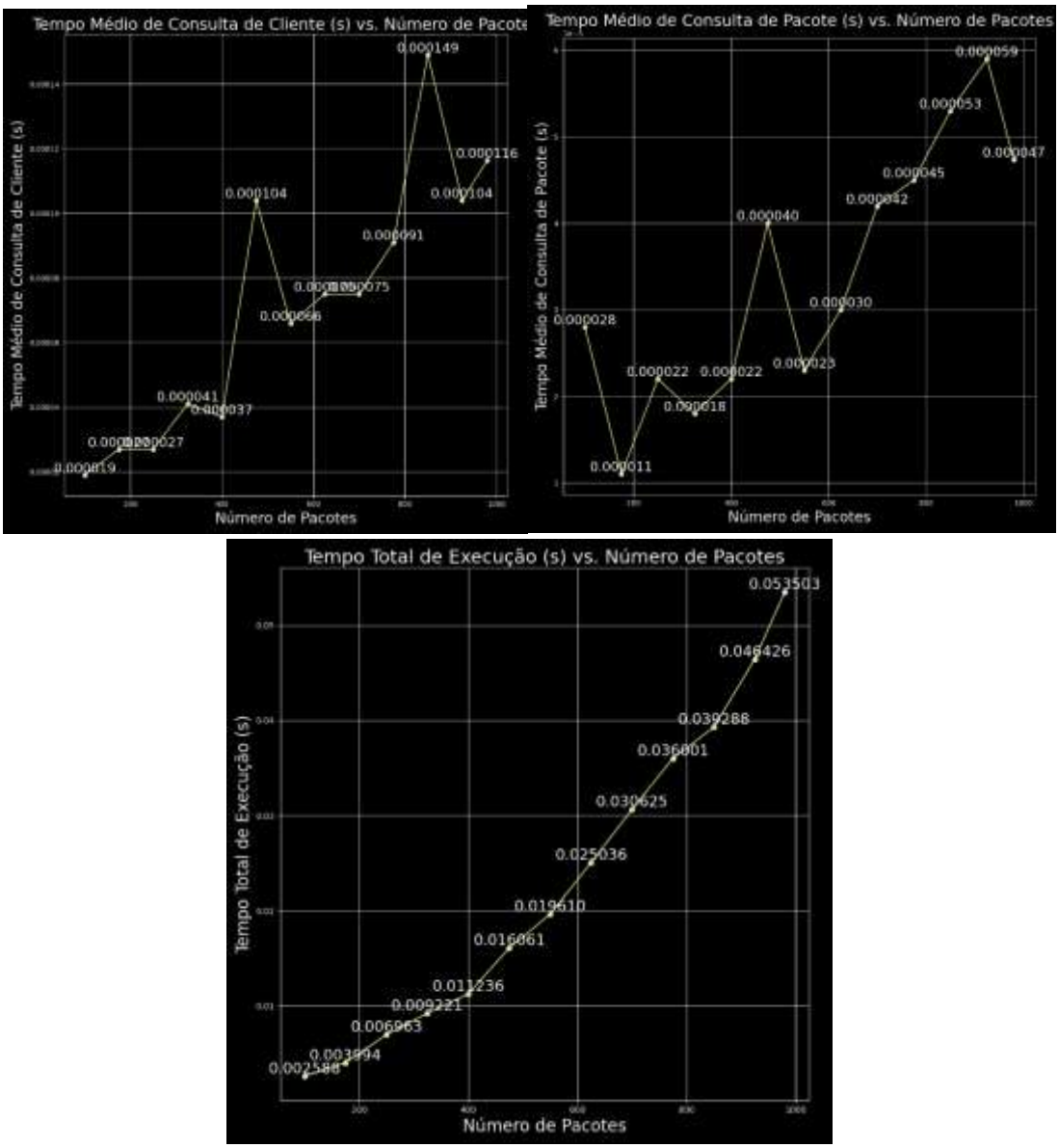

Figura 5. Gráficos dos tempos de execução total e médios relacionados ao número de pacotes.

5.2.4 Variação do número de clientes

A variação dessa dimensão não pareceu influenciar muito o tempo de execução total nem o tempo de execução médio para cada consulta. Os “spikes” no gráfico são o maior valor no eixo y para aquele número de clientes em testes que realmente influenciavam esses parâmetros, onde esse parâmetro se manteve com valor constante (mais especificamente, 50).

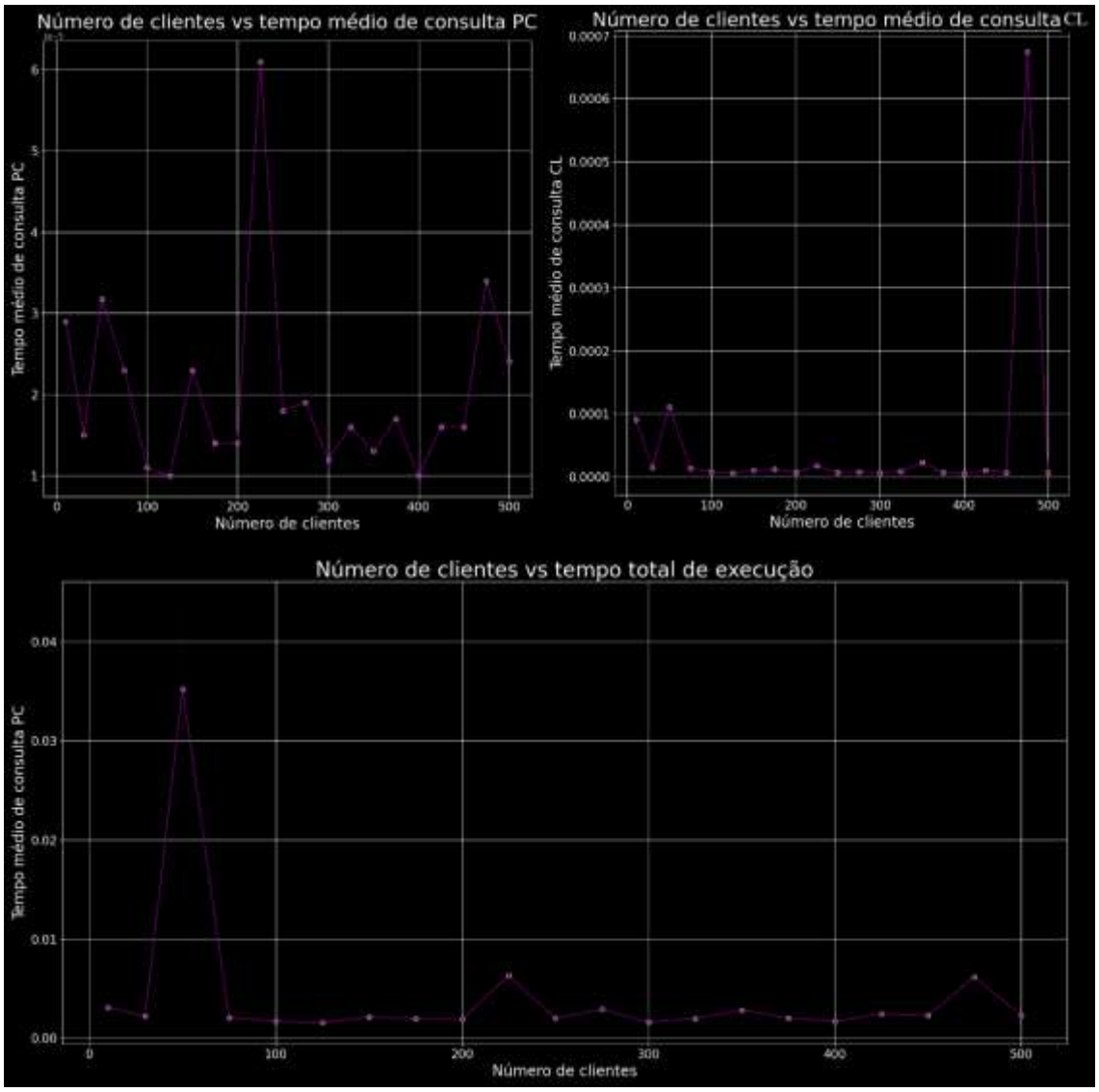

Figura 6. Gráficos com os tempos de execução total do programa e médios dos dois tipos de consulta

5.3 Conclusões

Alguns comportamentos anômalos foram observados, mas, no geral, pela análise nota-se que tudo ocorreu como o esperado. Os experimentos realizados demonstraram que o sistema é escalável e adaptável: responde bem a variações nos principais parâmetros da carga de trabalho (as dimensões número de eventos, de clientes e de pacotes).

6. Conclusão

Neste trabalho, um sistema eficiente para consultas no sistema logístico de transporte descrito foi implementado, no qual estruturas de dados como a árvore AVL foram decisivas para maximizar a eficiência delas quanto ao tempo de execução e a complexidade de espaço.

Durante o desenvolvimento, além do reforço de habilidades de programação, a habilidade de buscar sempre a eficiência de um programa pôde ser trabalhada. A experiência também destacou a importância do gerenciamento de memória, especialmente em sistemas com alocação dinâmica e exigência de robustez, aplicando práticas como a Rule of Three.

Bibliografia:

Meira, W. e Lacerda, A. (2025). Slides virtuais da disciplina de estruturas de dados.

Chaimowicz, L. and Macharet D. (2024). Slides virtuais da disciplina de Programação e Desenvolvimento de Software 2.

Disponibilizados via moodle. Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte.